DEPARTMENT OF COMPUTER APPLICATIONS
TRICHY-20

B.C.A
OBJECT ORIENTED PROGRAMMING WITH C++

M.BALAKRISHNAN MCA, M.Phil.
DEPARTMENT OF CA
JMC, TRICHY-20

UNIT III

Constructor and Destructor: Constructors - Parameterized constructor - Multiple constructor in a class -Dynamic initialization of the objects - Copy constructor - Dynamic constructor - Destructor. Operator Overloading and Type conversion: Defining operator overloading - Overloading unary operator - #Type conversion#

DYNAMIC INITIALIZATION OF OBJECT IN C++:

- Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during run time.
- It can be achieved by using constructors and by passing parameters to the constructors.
- This comes in really handy when there are multiple constructors of the same class with different inputs.

```
#include <iostream>
using namespace std;
class geeks
{
        int* ptr;

public:
        // Default constructor
        geeks()
        {
                // Dynamically initializing ptr
                // using new
```

```
                ptr = new int;
                *ptr = 10;
        }

        // Function to display the value
        // of ptr
        void display()
        {
                cout << *ptr << endl;
        }
};

int main()
{
        geeks obj1;

        // Function Call
        obj1.display();

        return 0;
}
```

Output:
10

COPY CONSTRUCTORS:
A copy constructor takes a reference to an object of the same class as itself as an argument.

```
class A
{
   A(A &x) //  copy constructor.
  {
     // copyconstructor.
  }
}
```

Example:

```
#include<iostream>
Using namespace std;
Class code
{
int id;
pubic:
code()
{   }
code(int a) {id=a;}
```

```
code(code &x)
{
id=x.id;
}
Void display()
{
cout<<id;
}
};
int main()
{
code A(100);
code B(A);
code C=A;
code D;
D=A;
cout<<"\n id of A: ";A.display();
cout<<"\n id of B: ";C.display();
cout<<"\n id of C: ";C.display();
cout<<"\n id of D: ";D.display();
return 0;
}
```

Output:

Id of A:100
Id of B:100
Id of C:100
Id of D:100

DYNAMIC CONSTRUCTOR:

- The constructor used for allocating the memory at runtime is known as the **dynamic constructor**.
- The memory is allocated at runtime using a **new** operator and similarly, memory is de allocated at runtime using the delete operator.

Example:

```
#include <iostream>
using namespace std;

class geeks
{
    const char* p;

public:
    // default constructor
    geeks()
```

```
    {
        // allocating memory at run time
        p = new char[6];
        p = "geeks";
    }
    void display()
    {
        cout << p << endl;
    }
};
int main()
{
    geeks obj;
    obj.display();
}
```

DESTRUCTORS:

A destructors, is used to destroy the objects than have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

```
~integer () { }
```

A destructor never takes any argument nor does it return any value.

Example:

```
#include< iostream>
using namespace std;
int count = 0;
Class alpha
{
public:
alpha ()
{
count++;
cout << "\n No. of object created "  < < count ;
}
~alpha ()
{
Cout << "\n  No. Of object destroyed "  < < count ;
Count--;
}
};
int main()
```

```
{
cout << "\n\n ENTER MAIN\n";
alpha A1, A2, A3, A4;
{
Cout << "\n\n ENTER BLOCK1\n";
alpha A5;
}
{
Cout << "\n\n ENTER BLOCK2\n;
Alpha A6;
}
cout << "\n\n RE-ENTER MAIN\n";
return 0;
}
```

OUTPUT:

```
ENTER MAIN
No. of object created 1
No .of object created 2
No. of object created 3
No. of object created 4

ENTER BLOCK1
No. of object created 5
No. of object destroyed 5

ENTER BLOCK2
No. of object created 5
No. of object destroyed 5

RE-ENTER MAIN
No. of object destroyed 4
No. of object destroyed 3
No. of object destroyed 2
No. of object destroyed 1
```

OPERATOR OVERLOADING:

- It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.
- Overloaded operator is used to perform operation on user-defined data type.
- For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.

The general form of an operator function is:

```
return type class name  :: operator op( arg list )
{
        Function body                   // task defined
 }
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is proceeded by the keyword operator. Operator op is the function name.

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.

Example:

```
#include <iostream>
using namespace std;
void add(int a, int b)
{
cout << "sum = " << (a + b);
}

void add(double a, double b)
{
        cout << endl << "sum = " << (a + b);
}

// Driver code
int main()
{
        add(10, 2);
        add(5.3, 6.2);
        return 0;
}
```

OVERLOADING UNARY OPERATORS:

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item.

```
#include<iostream.h>
using namespace std;
class space
{
Int x,y,z;
Public:
void getdata(int a,int b,int c);
Void display(void);
Void operator-();
};
Void space::getdata (int a,int b,int c)
{
X=a;
Y=b;
Z=c;
}
Void space :: display(void)
{
Cout<<x<<"\n";
Cout<<y<<"\n";
Cout<<z<<"\n";
}
void space::operator-()
{
X=-x;
Y=-y;
Z=-z;
}
Void main()
{
Space s;
s.getdata (10,20,30);
cout<<"s";
s.display();
-s;
Cout<<"s:";
S display();
```

}

OVERLOADING BINARY OPERATORS:

Binary operators can also be overloaded. Binary operators require two operands, and they are overloaded by using member functions and friend functions.

```
#include<iostream.h>
Using namespace std;
Class complex
{
Float x,y;
Public:
Complex () {};
Complex (float real, float img)
{
X=real;
Y=img;
}
Complex operator + (complex);
Void display(void);
};
Complex complex :: operator+(complex c)
{
Complex temp;
Temp x=x+c.x;
Tem y=y+c.y;
Return (temp);
}
Void complex :: display (void)
{
Cout<<x<<"+j"<<y<<"\n";
}
Void main()
{
```

```
Complex c1,c2,c3;
C1=complex (2.5,3.5);
C2=complex(1.6,2.7);
C3=c1+c2;
Cout<<"c1=";
C1 display();
Cout<<"c2=";
C2 display();
Cout<<"c3=";
C3 display();
}
```

TYPE CONVERSION:

A type cast is basically a conversion from one type to another. There are two types of type conversion.

1. Implicit Type Conversion

2. **Explicit Type Conversion**

IMPLICIT TYPE CONVERSION:

Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

An example of implicit conversion

```
#include <iostream>
using namespace std;
int main()
{
        int x = 10; // integer x
        char y = 'a'; // character c
        // y implicitly converted to int. ASCII
        // value of 'a' is 97
        x = x + y;
        // x is implicitly converted to float
        float z = x + 1.0;
        cout << "x = " << x << endl
```

```
                << "y = " << y << endl
                << "z = " << z << endl;
        return 0;
}
```

EXPLICIT TYPE CONVERSION:

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

```
#include <iostream>
using namespace std;

int main()
{
        double x = 1.2;
        // Explicit conversion from double to int
        int sum = (int)x + 1;
        cout << "Sum = " << sum;
        return 0;
}
```

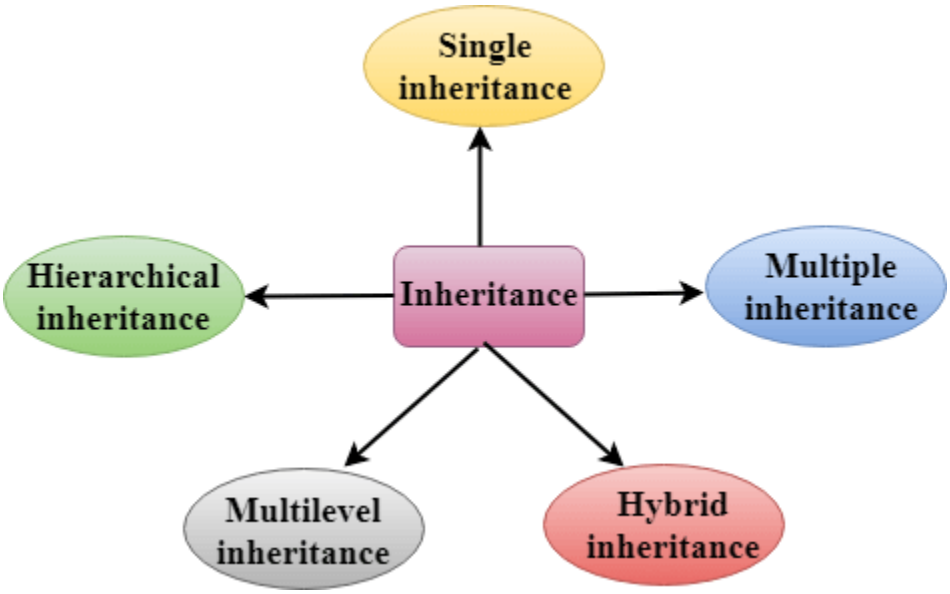
UNIT IV

Inheritance: Introduction - Single Inheritance - Multilevel inheritance - Multiple inheritance - Virtual base classes. Polymorphism: Pointers - Pointer to objects - this pointer - Pointer to derived classes - #Virtual Functions#

INHERITANCE INTRODUCTION

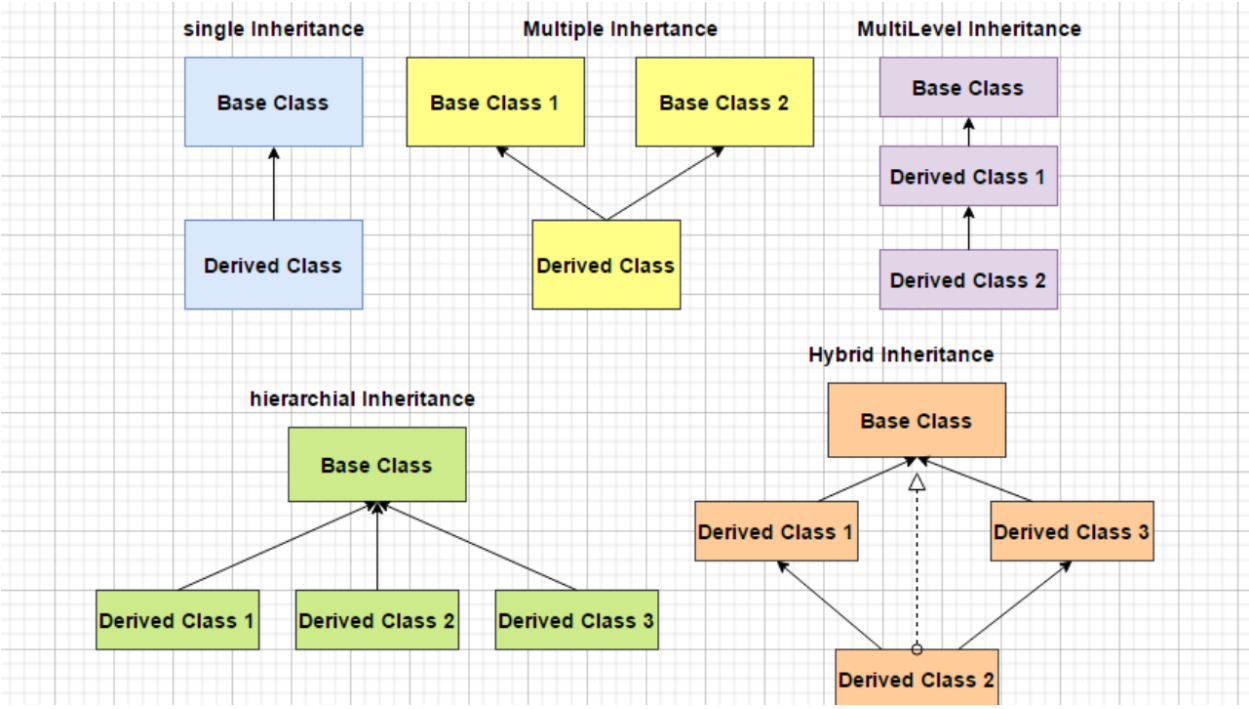
Reusability is yet another important feature of OOP. The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and the new one is called the derived class or subclass.

- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.



Types of Inheritance:

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hybrid inheritance
- Hierarchical inheritance



Defining Derived Classes

The general form of defining derived class is:

class derived-class-name: visibility-mode base-class-name

{
...../
...../ members of derived class
...../
};

The colon indicate that the derived class name is derived from the base-class-name. The visibility mode is optional and, if present, may be either private or public. The default visibility-mode is private.

Eg:

class ABC: private XYZ //private derivation

{

members of ABC

};

class ABC: public XYZ //public derivation

{

members of ABC

};

class ABC: XYZ //private derivation by default

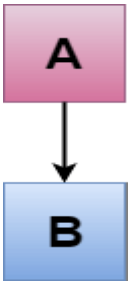
{

members of ABC

};

SINGLE INHERITANCE:

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



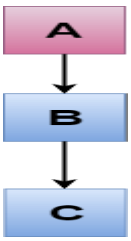
program:

```
#include<iostream>
using namespace std;
class student
{
protected:
char sex, name[20];
int rollno, weight, height;
public:
void getdata(void)
{
cout<<"\nEnter the name:";
cin>>name;
cout<<"\nEnter the rollno:";
cin>>rollno;
cout<<"\nEnter the sex(m/f):";
cin>>sex;
cout<<"\nEnter the height:";
cin>>height;
cout<<"\nEnter the weight:";
```

```
cin>>weight;
cout<<weight;
}
};
class stud: public student
{
public:
void putdata(void)
{
cout<<"\nName:"<<name;
cout<<"\nRollno:"<<rollno;
cout<<"\nSex:"<<sex;
cout<<"\nHeight:"<<height;
cout<<"\nWeight:"<<weight;
}
};
int main()
{
stud s;
cout<<"\n\t\t Single Inheritance\n";
cout<<"\n\t\t Enter the student details\n";
cout<<"\n\t\tDetails of student no:";
s.getdata();
cout<<end l;
s.putdata();
}
```

MULTILEVEL INHERITANCE

Multilevel inheritance is a process of deriving a class from another derived class.



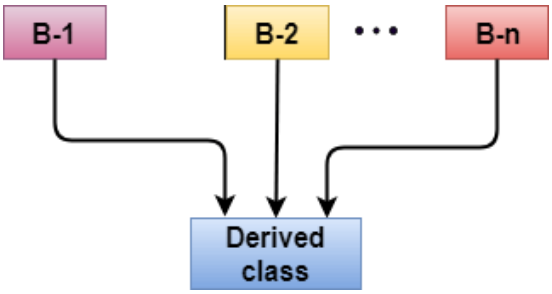
Example:

```
#include <iostream>
using namespace std;
class Animal
{
public:
void eat()
        {
        cout<<"Eating..."<<endl;
        }
};
class Dog: public Animal
{
public:
void bark()
{
cout<<"Barking..."<<endl;
}
};
class BabyDog: public Dog
{
public:
void weep()
 {
cout<<"Weeping...";
}
};
int main(void)
{
babydog d1;
d1.eat();
d1.bark();
```

```
d1.weep();
return 0;
}
```

MULTIPLE INHERITANCE:

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

```
class D:visibility B-1,visibility B-2...
{
...
....(Body of D)
....
};
```

Example:

```
#include <iostream>
using namespace std;
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   }
};

class B
{
   protected:
   int b;
   public:
```

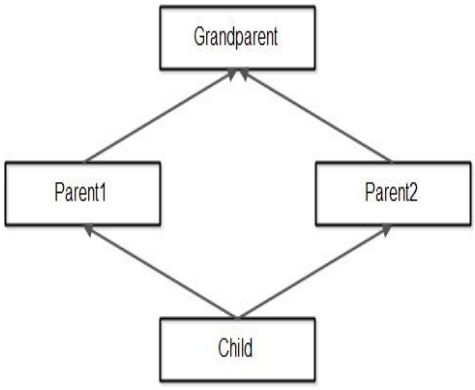
```
    void get_b(int n)
    {
       b = n;
    }
};
class C : public A,public B
{
  public:
    void display()
    {
       std::cout << "The value of a is : " <<a<< std::endl;
       std::cout << "The value of b is : " <<b<< std::endl;
       cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();

   return 0;
}
```

VIRTUAL BASE CLASS:

An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.

C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.



Example:

```
class A
{
  public:
```

```
    int i;
};

class B : virtual public A
{
  public:
    int j;
};

class C: virtual public A
{
  public:
    int k;
};

class D: public B, public C
{
  public:
    int sum;
};

int main()
{
  D ob;
  ob.i = 10; //unambiguous since only one copy of i is inherited.
  ob.j = 20;
  ob.k = 30;
  ob.sum = ob.i + ob.j + ob.k;
  cout << "Value of i is : "<< ob.i<<"\n";
  cout << "Value of j is : "<< ob.j<<"\n";
  cout << "Value of k is :"<< ob.k<<"\n";
  cout << "Sum is : "<< ob.sum <<"\n";

  return 0;
}
```

POLYMORPHISM:

The word "polymorphism" means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

A real-life example of polymorphism is a person who at the same time can have different characteristics. **A man at the same time is a father, a husband, and an employee.**

Types of Polymorphism

- **Compile-time Polymorphism.**
- **Runtime Polymorphism.**

POINTERS

A pointer is a derived data type that refers to another data variable by storing the variables memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Declaring and Initializing Pointers

data-type *pointer-variable;

Program:

```
#include<iostream.h>

#include<conio.h>

void main()

{

int a, **ptr1, **ptr2;

clrscr();

ptr1=&a;

ptr2=&ptr1;

cout<<"The address of a:"<<ptr1<<"\n";

cout<<"The address of ptr1:"<<ptr2;

cout<<"\n\n";

cout<<"After increamenting the address values :\n\n";

ptr1+=2;

cout<<"The address of a :"<<ptr1<<"\n";

ptr2+=2;

cout<<"The address of ptr1:"<<ptr2<<"\n";

}
```

POINTERS TO OBJECTS

A pointer can point to an object created by a class.

Program:

```
#include<iostream.h>
using namespace std;
class item
{
int code;
float price;
```

```
public:
void getdata(int a,float b)
{
code=a;
price=b;
}
void show(void)
{
cout<<"code :"<<code<<"\n";
cout<<"price :"<<price<<"\n";
}
};
const int size=2;
int main()
{
item *p=new item[size];
item *d=p;
int x,i;
float y;
for(i=0;i<size;i++)
{
cout<<"Input code and price for item"<<i+1;
cin>>x>>y;
p->getdata(x,y);
p++;
}
for(i=0;i<size;i++)
{
cout<<"Item:"<<i+1<<"\n";
d->show();
d++;
}
return 0;
}
```

Output:

```
Input code and price for item1 40 500
Input code and price for item2 50 600

Item:1
code:40
price:500

Item:2
```

code:50
price:600

this POINTER

C++ uses a unique keyword called this to represent an object that invokes a member function. This is a pointer that points to the object for which this function was called.

There can be 3 main usage of this keyword in C++.

- It can be used to pass current object as a parameter to another method.
- It can be used to refer current class instance variable.
- It can be used to declare indexers.

Example:

```
#include <iostream>
using namespace std;
class employee
{
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     employee(int id, string name, float salary)
      {
          this->id = id;
         this->name = name;
         this->salary = salary;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
   e1.display();
   e2.display();
   return 0;
}
```

POINTERS TO DERIVED CLASSES

C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

```
B *cptr;        // here is pointer to class B type variable
B b;            // base object
D d;            // derived object
cptr=&b;        // cptr points to object b
```

VIRTUAL FUNCTIONS

A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at runtime.

Rules for Virtual Functions

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

```
#include<iostream>
using namespace std;

class base
{
public:
   virtual void print()
   {
      cout << "print base class\n";
   }

   void show()
   {
      cout << "show base class\n";
```

```
    }
};

class derived : public base
 {
public:
    void print()
    {
        cout << "print derived class\n";
    }

    void show()
    {
        cout << "show derived class\n";
    }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
    return 0;
}
```

Output:
print derived class
show base class

UNIT V

Working with Files: Introduction - Classes for File stream - Opening and closing the file - Detecting end of file – File modes. Templates: Introduction - Class templates - Class templates with multiple parameters - Function templates

FILES INTRODUCTION:

The information / data stored under a specific name on a storage device, is called a file.

C++ Files:

The fstream library allows us to work with files.

To use the fstream library, include both the standard <iostream> **AND** the <fstream> header file:

```
#include <iostream>
#include <fstream>
```

Stream:

It refers to a sequence of bytes.

Text file:

It is a file that stores information in ASCII characters. In text files, each line of text is terminated with a special character known as EOL (End of Line) character or delimiter character. When this EOL character is read or written, certain internal translations take place.

Binary file:

It is a file that contains information in the same format as it is held in memory. In binary files, no delimiters are used for a line and no translations occur here.

CLASSES FOR FILE STREAM OPERATION:

ofstream:

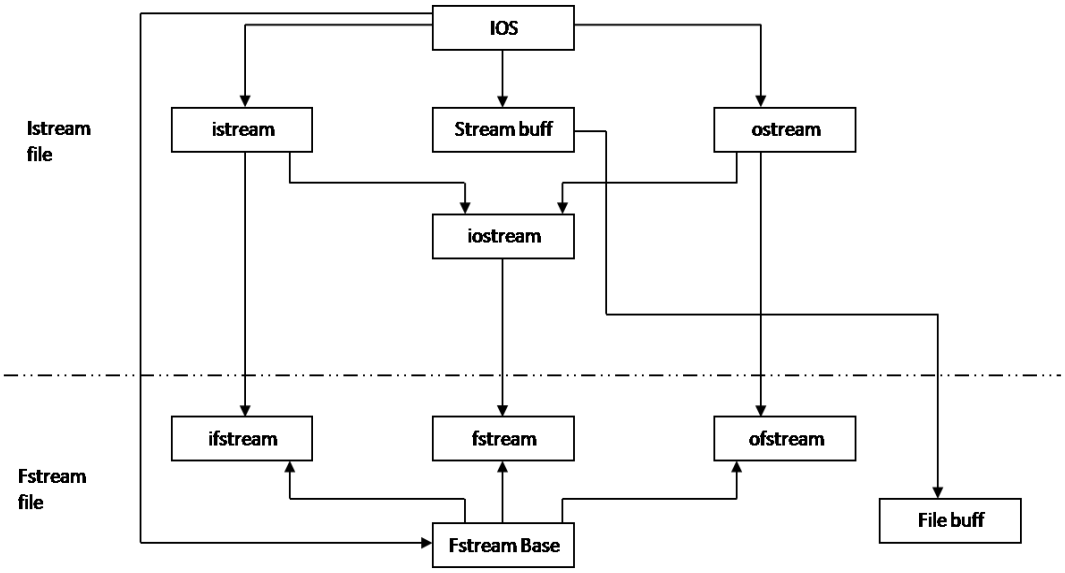
Stream class to write on files

ifstream:

Stream class to read from files

fstream:

Stream class to both read and write from/to files.



OPENING AND CLOSING THE FILE:

- A name for the file.
- Data type and structure of the file.
- Purpose (reading, writing data).
- Opening method.
- Closing the file after use.

FILE STREAM CLASSES

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes.Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to file streams.Serves as a base for **fstream,ifstream** and **ofstream** class.Contains **open()** and **close()** functions.
ifstream	Provides input operations.Contains **open()** with default input mode.Inherits the functions **get(),getline(),read(),seekg(),tellg()** functions from istream.
ofstream	Provides output operations.Contains **open()** with default output mode.Inherits **put(),seekp(),tellp()** and **write()** functions from **ostream**.
fstream	Provides support for simultaneous input and output operations.Contains open with default input mode.Inherits all the functions from **istream** and **ostream** classes through **iostream**.

OPENING A FILE

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing or ifstream object is used to open a file for reading purpose only.

open (file_name, mode);

void open(const char *filename, ios::openmode mode);

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
  fstream file;
  file.open ("abc.txt", ios::out | ios::in );
  return 0;
}
```

CLOSING A FILE

A C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files.

- flushes the streams
- releases the allocated memory
- Closes opened files.

void close();

Example:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  ofstream file;
  file.open ("abc.txt");
  file.close();
  return 0;

}
```

Writing to a file

A C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a file

A read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

DETECTING END OF FILE:

The **eof()** method of **ios class** in C++ is used to check if the stream is has raised any EOF (End Of File) error. It means that this function will check if this stream has its eofbit set.

Syntax:
bool eof() const;

Parameters: This method does not accept any parameter.
Return Value: This method returns true if the stream has eofbit set, else false.

FILE MODES:

Parameter	Meaning
ios::app	causes all output to the file to be appended to the end
ios::ate	causes a seek to the end of the file occur when file is opened
ios::binary	causes a file to be opened in binary mode
ios::out	specifies that the file is capable of output
ios::in	specifies that the file is capable of input
ios::nocreate	causes open() to fail if the file doe not already exist.
ios::noreplace	causes open() to fail if the file already exist
ios::trunk	causes the contents of a preexisting file by the same name to be destroyed and truncate the file to zero length

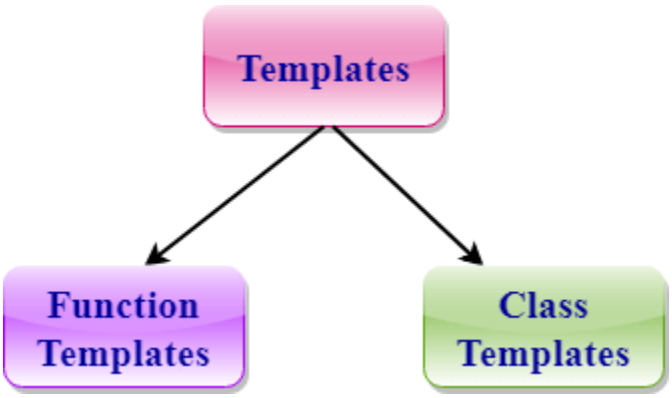
TEMPLATES:
INTRODUCTION:

A **template** is a simple yet very powerful tool in C++.
It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by the keyword 'class'.

Templates can be represented in two ways:

- Function templates
- Class templates



CLASS TEMPLATE

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

class_name<type> ob;

Example:

```
#include <iostream>
using namespace std;
template<class T>
class A
{
```

```
   public:
   T num1 = 5;
   T num2 = 6;
   void add()
   {
      std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
   }

};

int main()
{
   A<int> d;
   d.add();
   return 0;
}
```

Output:

Addition of num1 and num2 : 11

FUNCTION TEMPLATE

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
   // body of function.
}
```

Example:

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
   T result = a+b;
   return result;
```

```
}
int main()
{
  int i =2;
  int j =3;
  float m = 2.3;
  float n = 1.2;
  cout<<"Addition of i and j is :"<<add(i,j);
  cout<<'\n';
  cout<<"Addition of m and n is :"<<add(m,n);
  return 0;
}
```

Output:

Addition of i and j is :5
Addition of m and n is :3.5

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
   // Body of the class.
} ;
```

Example:

```
#include <iostream>
     using namespace std;
     template<class T1, class T2>
    class A
    {
        T1 a;
        T2 b;
        public:
       A(T1 x,T2 y)
      {
         a = x;
         b = y;
       }
          void display()
         {
```

```
            std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
        }
    };

    int main()
    {
        A<int,float> d(5,6.5);
        d.display();
        return 0;
    }
```

Output:

Values of a and b are : 5,6.5